

AVERAGING AN ARRAY

The array STOCK contains a list of 1000 whole numbers (integers). The following pseudocode counts how many of these numbers are non-zero, adds up all those numbers, and then prints the average of all the non-zero numbers (divides by COUNT rather than dividing by 1000).

```
COUNT = 0
TOTAL = 0

loop N from 0 to 999
    if STOCK[N] > 0 then
        COUNT = COUNT + 1
        TOTAL = TOTAL + STOCK[N]
    end if
end loop

if not(COUNT = 0) then
    AVERAGE = TOTAL / COUNT
    output "Average = " , AVERAGE
else
    output "There are no non-zero values"
end if
```

~~~~~ Comparing Numbers

The operators for comparing numbers are: = , < , > , <= , >=

There is no operator for "not equal". When "not equal" is needed, it will be written:

```
if not(COUNT = 0) then ...
```

Output

The **output** command displays text output that a user could read. The formatting (e.g. spaces) and exact output device (printer or screen) are unimportant. Students may assume that the output appears on a display screen. Each subsequent output command displays text on a new line. Several items can be printed on a single line by separating them with commas. Presumably blank spaces are printed between items in the output list.

Division Operator

The division / operator always calculates the correct decimal value - it does not truncate or round-off. For integer division, the **div** operator must be used.

Counting Loops

The example contains a "counting loop", with a variable N that counts from 0 to 999.

This is similar to a **for** loop in BASIC or C++ or Java. Counting loops are very simple and will always count by 1 from the first value to the ending value inclusive. If the ending value is smaller than the starting value, the variable will count down by -1. The counting step happens at the end of the loop. If the counting (index) variable is accessed after the loop, it equals the ending value. There is no command for ending (breaking) a loop before reaching the end value.

COPYING FROM A COLLECTION INTO AN ARRAY

The following code reads all the names from a NAMES collection and copies them into a LIST array, but eliminating duplicates. That means each name is checked against the names that are already in the array. The collection and the array are passed as parameters to the method.

```
COUNT = 0      // number of names currently in LIST

loop while NAMES.hasNext()

    DATA = NAMES.getNext()

    FOUND = false
    loop POS from 0 to COUNT-1
        if DATA = LIST[POS] then
            FOUND = true
        end if
    end loop

    if FOUND = false then
        LIST[COUNT] = DATA
        COUNT = COUNT + 1
    end if
end loop
```

~~~~~

### Sub-programs (methods and functions)

IB pseudocode has no representation for methods, functions or other sub-program constructs. Hence, all pseudocode in exams simply represents the contents of one code block. Exam questions may specify the existence of a new pre-defined function, presumably available from an unspecified library, that may be used in the current problem. But students will not be expected to read, trace or otherwise understand the actual contents of such functions.

### Variables, Parameters and Return Values

IB pseudocode does not specify variable types. All variables are "global" for the code presented. Hence the variable FOUND can be used throughout the algorithm, regardless of where it first appears. Of course, its value would not be used before a value has been assigned.

Variables are typeless and can contain any value - numbers (integer or floating point), booleans or Strings. Casting and other type-conversion operations do not exist and should not be assumed. For example, using **mod** with a floating point value is undefined and inappropriate.

Since there are no sub-program constructs, there are also no parameters or return values - except when using a "pre-defined" function - the appropriate use of parameters and return values will then be explained in the question.

### Length of an Array

There is no command for determining the length of an array. The length of an existing array may be specified in the question. Otherwise, the length is unspecified and hence unlimited. Arrays are zero-based, so ARRAY[0] always exists. However, a question might specify that location 0 is not used.

## FACTORS

Print all the factors of an integer. Prints 2 factors at a time, stopping at the square-root. It also counts and displays the total number of factors.

```
// recall that
// 30 div 7 = 4
// 30 mod 7 = 2

NUM = 140
F = 1
FACTORS = 0

loop while F*F <= NUM

    if NUM mod F = 0 then

        D = NUM div F
        output NUM , " = " , F , "*" , D

        if F = 1 then
            FACTORS = FACTORS + 0
        else if F = D then
            FACTORS = FACTORS + 1
        else
            FACTORS = FACTORS + 2
        end if

    end if

    F = F + 1

end loop

output NUM , " has " , FACTORS , " factors "
```

~~~~~

Indentation

In exam questions, pseudo-code will always be properly indented, so that matching constructs can be easily identified - e.g. loop / end loop , if / end if. This is especially important for "nested" constructs.

Recall

Sometimes exam pseudo-code will contain a reminder of code elements that are not used very often, or that might be otherwise unfamiliar or confusing for some students. However, students and teachers should not assume that such hints will always be available. Teachers need to familiarize students with all the standard pseudo-code constructs.

TOP TEN SCORES

Assume that NAMES is an array containing the names of 10 players. A parallel array SCORES contains their scores in a game. The SCORES are in descending order (highest to lowest). NAMES[0] contains the name of the player with the top score. NAMES[9] contains the name of the player with the lowest score. The following algorithm inserts a new name and score into the correct place in the list if the new score is higher than the lowest existing score. It must move all the lower scores down one position in the list, eliminating the lowest score.

```
NAMES = [ "Hi", "Fy", "Di", "Ed", "Al", "Go", "Ma", "Mi", "Ha", "Lo" ]
SCORES =[ 900, 800, 700, 600, 550, 499, 450, 425, 390, 123]

PLAYER = "Nu"
POINTS = 500

P = 0
loop while (P < 10) and (SCORES[P] > POINTS)
    P = P + 1
end loop

if P < 9 then
    loop X from P to 8
        NAMES[X+1] = NAMES[X]
        SCORES[X+1] = SCORES[X]
    end loop
end if

if P <= 9 then
    NAMES[P] = PLAYER
    SCORES[P] = POINTS
end if
```

~~~~~

## Creating Arrays

The first two lines create arrays and specify starting contents. Many exam questions will not specify the starting contents of arrays, but rather **assume** that arrays already exist with unspecified contents.

## Boolean Operators

For if..then.. decisions and while.. loops, the Boolean operators "and", "or", and "not" are written as words, NOT using the symbols from Java and C++ (&&, ||, !). Examiners will use parentheses if needed for clarity in Boolean expressions (as in the while.. command above).

## Assignment Statements

The command, **SCORES[X+1] = SCORES[X]**, copies data from position X into position X+1 - that is, the assignment statement moves data from the right side to the left side, like Java, C++ and Basic and many other languages.

## **Errors**

The purpose of pseudocode is to outline successful algorithms. Errors are not an issue. Hence, overflows and underflows do not occur. Type mismatches do not occur (e.g. do not "crash the program"), although students and examiners need to code algorithms that CAN function. For example, they cannot assume that a variable simultaneously contains two different values, or assume that it is possible to compare "seven" and "VII" and 7 successfully. And they do need to write correct "syntax" in the sense that every **if..** must have a matching **end if**, every **loop** has a matching **end loop**, mathematical expressions are written correctly, variable names are spelled the same throughout the code, etc.